Int J Adv Manuf Technol (2017) 92:1579–1590
DOI 10.1007/s00170-017-0154-z

ORIGINAL ARTICLE

Agent-based middleware architecture for reconfigurable manufacturing systems

Rafael Priego[1] · **Nagore Iriondo**[1] · **Unai Gangoiti**[1] · **Marga Marcos**[1]

Received: 2 August 2016 / Accepted: 9 February 2017 / Published online: 15 March 2017

Abstract Modern manufacturing systems are expected to be flexible and efficient in order to cope with challenging market demands. Thus, they must be flexible enough as to meet changing requirements such as changes in production, energy efficiency, performance optimization, fault tolerance to process or controller faults, among others. Demanding requirements can be defined as a set of quality of service (QoS) requirements to be met. This paper proposes a generic and customizable multi-agent architecture that, making use of distributed agents, monitors QoS, triggering, if needed, a reconfiguration of the control system to recover QoS. As a proof of concept, the architecture has been implemented to provide availability of the control system understood as service continuity. The prototype has been tested in a case study consisting of an assembly cell where assessment of the approach has been conducted.

Keywords Multi-agent systems · Middleware · Quality of service · Control system availability · Dynamic reconfiguration

✉ Rafael Priego
rafael.priego@ehu.eus

Nagore Iriondo
nagore.iriondo@ehu.eus

Unai Gangoiti
unai.gangoiti@ehu.eus

Marga Marcos
marga.marcos@ehu.eus

[1] Department of Automatic Control and System Engineering, University of the Basque Country, C/Alameda Urquijo s/n, 48013 Bilbao, Spain

1 Introduction

In recent years, there has been an increase on the investment effort from public institutions to reinforce or recover the manufacturing industries. This sector offers a suitable opportunity for increasing innovation, economic growth and job creation. Initiatives such as Factory of the Future in the European Union [1], the Industrie 4.0 initiative driven by the German Federal Government [2] and the so-called Advanced Manufacturing launched by the US Government [3] are clear examples. All these initiatives pursue the implementation of high-tech manufacturing processes based on the use of adaptive and smart manufacturing equipment and systems, aiming at automating, controlling and optimizing the processes, ensuring plant availability while providing high quality production with zero defects.

The achievement of these objectives requires automation production system to exhibit the ability to self-reconfigure to meet quality of service (QoS) requirements. Reconfiguration mechanisms provide the system with the ability of switching from one configuration to another, improving the efficiency of the system with respect to sudden changes on customer demands and/or unpredictable events, like failure or disruptions.

Different works can be found in the literature dealing with flexibility in manufacturing systems. Some of these works deal with the use of reconfiguration mechanisms at production level by implementing different mechanisms to optimize the production process. For instance, Morenas, Higuera and Alonso [4] assign priority to products based on the particular customer, the deadline of the order and the order of arrival. On the other hand, Nouri [5] uses other parameters such as machine workload, material handling, operational time and operation sequence of the parts, among others. Other works uses the re-scheduling to reduce global energy consumption [6] or to deal with machine or actuator failures [7, 8].

Other focus of research is the dynamic reconfiguration of the control system. Botygin and Tartakovsky [9] optimize the workload of the system controllers following a dynamic ranking table. In a similar way, Binotto et al. [10] balance the task of each controller based on order arrival time and the current workload of the controllers. In [11], the goal of the workload balance is to optimize energy efficiency. Reconfiguration has also used in order to add fault tolerance to the automation system, either to controller faults [12] or to network failures [13].

Thus, QoS is understood in different ways, from optimizing the manufacturing orders to meet order requirements (deadlines, customer, etc.) to optimizing production efficiency, energy efficiency or response to process or controller faults. But they have in common the reconfiguration of the automation system. All these works aim at ensuring a specific QoS (production optimization, process fault tolerance, controller failure tolerance, workload balance, among others) by offering a custom solution to the concrete issue. This paper contributes a generic architecture identifying the key components to deal with QoS loss detection and reaction that can be customized for different QoS goals including the ones commented above.

It generalizes and formalizes previous works of authors. [14] presents a preliminary work on a multi-agent system (MAS) which provides fault tolerance to controller faults, recovering the whole execution in other controller. [15] introduces the concept of Mechatronic Component (MC) as the code in charge of controlling a part of the process, and it analyzes when it is possible to recover its functionality after a controller fault. In [16], the concept of backup MC for state tracking is proposed. The novel contributions of this work go beyond them as follows:

- It formulates the MC concept giving guidelines to the developer to define it, identifying the non-recoverable execution states.
- It proposes a MAS-based implementation of the complete system architecture, identifying and defining different types of agents with specific roles.
- It defines code templates for MCs.

The paper also presents, as a proof of concept, a middleware prototype that implements the control system availability under controller failures, recovering the set of MCs of failed controllers in others. The proposed generic architecture can be customized for other QoS, preventing that they can be measured from the state of the automation system, such as the QoS definition analysed in the literature (process faults, production efficiency, etc.).

The remainder of the paper is as follows: Section 2 presents related work focusing on how the reconfiguration mechanisms are implemented. Section 3 formulates the concept of mechatronic component extending previous definitions in the literature to add the possible execution states. These latter inform about when the state of the MC can be derived from the current automation system state and, thus, reconfiguration actions can be performed. Section 4 presents the agent-based implementation of the runtime platform, i.e. the middleware components that, making use of an ontology proposal, implement QoS monitoring and QoS recovering. Section 5 is dedicated to a case study where illustrative examples are assessed. Finally, Section 6 outlines the most important conclusions and future work.

2 Related work

This section analyses the different technologies used for implementing reconfiguration mechanisms in manufacturing. Most of the works found use the IEC61499 standard [17]. Unfortunately, its consolidation in industry has not been as expected, being the IEC61131-3 standard [18] the most used in industrial applications. Notwithstanding this, it is worthy to look at how the reconfiguration mechanisms are implemented.

Olsen et al. [19] propose the reconfiguration of IEC61499 control systems, by means of adding, removing or reconnecting the functional blocks (FB) of the application. The reconfiguration actions to cope with the different situations, like machines or controller failures, must be previously defined by the user.

Khalgui and Mosbahi [20] extend the concept of adding and removing FB in order to allow the relocation of FBs into other controllers. This relocation is based on moving the code of the FB as well as its execution data (its state). This type of reconfiguration is known as *stateful reconfiguration*. The authors propose an external multi-agent system that provides the relocation of the FBs. In a similar way, the so-called eCEDAC approach [21] implements the reconfiguration through the so-called reconfiguration execution control function block, which is able to generate a copy of an FB in other controller, as well as restoring their data, rearrange their communications and remove the old FB. On the other hand, Yan and Vyatkin [22] propose two new types of FB: the agent FB and the transfer FB in charge of performing the reconfiguration. However, the application is responsible for performing the monitoring and detection of failures and, making use of the mechanisms provided, reconfiguring the control system.

In [23, 13], FBs are duplicated in different controllers. One of the controllers acts as a master being in charge of monitoring controller and/or network failures, respectively, deciding when a duplicated FB must be activated from the last execution state.

Although, in other application areas, more generic architectures that manage concurrently multiple types of QoS have been proposed, for instance in control systems [24], and healthcare [25], as far as authors know, solutions proposed for manufacturing systems only tackle specific type of QoS

or offer mechanisms to be handled by the application. Besides, the implementation of such strategies in current IEC61131-3 automation systems is not straightforward.

On the other hand, when a failure is detected, it is necessary to determine the actions to be done on the production system before and after the reconfiguration takes place to prevent inconsistency states. Lepuschitz et al. [7] studies this problem under process instrumentation failures.

The present work takes these ideas and goes beyond integrating them into a generic architecture that provides QoS management achieving, thus, production unaware reconfiguration.

3 Formulation of the mechatronic component concept

The concept of mechatronic component (MC) was first proposed by Thrambouilidis [26] as a component having a mechanical part, an electronic part and a software part. Being a mechatronic system, it is a network of mechatronic components that collaborate through appropriate links to achieve system objectives. Lüder et al. [27] define mechatronical units as containers of relevant information required within the engineering of manufacturing systems (mechanical data, functional data, control data, economical data, technical data and others). In this work, the concept is extended to represent runtime information: In particular, the situations in which the current state of the mechatronic component can be derived from the current values of the software variables.

This information is fundamental to enable dynamic reconfiguration. In the case of using dynamic reconfiguration for assuring some type of QoS (such as, for instance, system availability or efficiency), it is necessary not only to monitor such QoS (using specific techniques such as for instance heart beat or work load, respectively, in all controllers) but it is also necessary to assure that the reconfiguration is possible and the manufacturing process will not suffer unpredictable effects. For instance, if the work of controllers is unbalanced and it is necessary to transfer control tasks among controllers, the transference of workload must be done at instant in which the production process will not be affected. This means that the operation state of the process must be known by the controller.

Depending on the manufacturing process and the design of the automation production system, it is possible to distinguish the following runtime situations [14]:

An operation state of the MC is *non-critical* if it may be activated directly in another controller (after de-activation in the current controller or after a controller failure) having as initial state the last known state of the MC. On the contrary, *critical* states are those that are not fully represented by the current values of the software variables of the MC, preventing activation/de-activation of the MC in order to avoid unpredictable process behaviour.

Table 1 summarizes when a MC reconfiguration, aiming at recovering some QoS, can be launched attending at the MC state. Note that recovering a MC from a controller failure is a special case in which a critical state has two variants: those cases in which even the state is not completely known, the MC could be re-started from a checkpoint state, and those in which the state is unpredictable, and thus, the MC cannot be re-started.

This extension of the MC concept leads to the following formulation. The overall control system is composed by a set of MCs that in turn is a set of elements containing relevant information about the operating part of the system:

$$\begin{aligned} \mathrm{Sys} &= \{\mathrm{MC}_i\}, \quad i = 1 \ldots n \\ \mathrm{MC}_i &= \{\mathrm{types}_i,\ \mathrm{main}_i, S_i, \mathrm{VI}_i, \mathrm{VO}_i,\ \Sigma_i,\ \Psi_i,\ \delta_i\}, \end{aligned} \tag{1}$$

where:

- $\mathrm{types}_i = \{D_i,\ P_i\}$ represents the set of data types (D_i) and program organization unit (POU in the context of the IEC 61331-3 software model) types (P_i) that compose the control logic of the MC_i.
- $\mathrm{main}_i \in \mathrm{types}_i$ is the POU type of the main program from which the rest of POU types of the MC_i are instantiated and used.
- S_i is the set of variables corresponding to the MC_i. The variables are characterized by their name and type, and in the case of inputs and outputs, they are also characterized by its physical address (2).

$$\begin{aligned} s_i^j &= (\mathrm{name}, \mathrm{type}, [\mathrm{address}]);\ s_i^j \in S_i;\ i = 1 \ldots n\ \mathrm{MCs}, \\ j &= 1 \ldots m\ \text{state variables of } \mathrm{MC}_i \end{aligned} \tag{2}$$

- $\mathrm{VI_i} \subset \mathrm{S_i}$ is the set of inputs coming from the external world (process, operator, HMI, …) that correspond to global variables.
- $\mathrm{VO}_i \subset S_i$ is the set of outputs associated to the physical variables related to the physical part of the MC_i, corresponding to global variables.

The rest of elements that compose the MC define the critical states of the production system in which the MC must not be activated/de-activated as this action may lead to an inconsistent state of the MC. As commented above, they can be represented as the values of selected variables in S_i. Note that a controller failure in any of these states will lead to a non-direct recovery.

Critical intervals are defined as the set of critical states that have associated the same recovery method (checkpoint state and/or recovery actions). The critical interval is characterized by the following:

Table 1 MC reconfiguration

MC state	Reconfiguration (MC de-activation/activation)	Controller failure recovery
Non-critical	Always possible (from the last known state) (*reconfiguration enabled*)	Always possible (from the last known state). (*reconfiguration enabled*)
Critical state	Wait until non-critical state is reached (*reconfiguration disabled*)	MC can be launched from a previous known state (*checkpoint*), possibly after executing recovery actions. (*reconfiguration enabled*)
		MC recovery is not possible. Action: safe stop. Operator warning. (*reconfiguration disabled*)

i. An expression involving values of variables of the MC state.
ii. A recovery action and/or a recovery state (checkpoint) needed to resume the execution. Note that a state that does not belong to a critical interval can be directly recovered.

Therefore,

- $\Sigma_i = \{\Sigma_i^k\}, k = 1 \ldots l$ critical intervals, is the input alphabet that defines all the expressions associated to critical intervals of the set of MCs. Every critical interval has its own expression, Σ_i^k, that allows determining if an execution state belongs to the interval. It is a Boolean expression composed by arithmetic and logical operations of state values following the grammar:

$$
\begin{aligned}
&< \text{Exp} > :: = < \text{Var} >< \text{arithmetic} >< \text{Value} > \Big| < \text{Exp} >< \text{logic} >< \text{Exp} > \\
&< \text{arithmetic} > :: = \leq | \geq | < | > | = \\
&< \text{logic} > :: = \text{OR} \Big| AND \\
&< \text{Var} > \in S \\
&< \text{Value} > \in \mathbb{R}
\end{aligned}
$$

- $\Psi_i = \{\Psi_i^k\} |\ \Psi_i^k = \{\xi_i^k, \lambda_i^k\}; k = 1 \ldots l$ critical intervals, represents the output alphabet that defines all possible recovery methods. The recovery methods for critical states include an action ($\xi_i^k \in \text{types}_i$), a checkpoint ($\lambda_i^k \subset S_i$) or both. The checkpoint is the state value from which the execution must be resumed.

$\delta_i : S_i \times \Sigma_i \rightarrow \Psi_i$ is the output function that assigns to a critical interval an element of the output alphabet.

4 MAS architecture for QoS management

This section describes the main concepts of the architecture for QoS management, its components as well as the mechanisms for QoS monitoring, QoS loss detection and QoS recovering if possible making use of the MC concept proposed in Section 3.

The architecture extends JADE (Java Agent DEvelopment Framework) [28, 29] with a set of agents that allow managing the whole system QoS requirements for manufacturing control applications. JADE is a software framework that aids developers to build agent applications in compliance with FIPA (Foundation for Intelligent Physical Agents) specifications [30] for interoperable intelligent multi-agent systems. The purpose of JADE is to simplify development while ensuring standard compliance through a comprehensive set of system services and agents.

JADE includes all the system agents specified by FIPA that manage the platform illustrated at the bottom part of Fig. 1:

- The *Agent Management System*, responsible for agent creation, removal and migration mechanisms.
- The *Agent Communication Channel*, in charge of interoperability within and among different platforms.
- The *Directory Facilitator* that supplies a yellow page service to the agent platform where agents can register services or look for required services.

The proposed architecture adds four types of agents (top part of Fig. 1). Some of them are part of the basic architecture while there are agents for managing the different QoS to be assured. Finally, there are a set of agents that depends on the particular application in charge of collecting information from the current operation state of the automation system.

- The Middleware Manager (MM) agent is unique on the system (although it can be redundant using the services provided by JADE. This extension is out of the scope of this work). It is the main orchestrator and manages the System Repository (SR), containing dynamic information about current state of the automation system (controllers and MCs).
- The QoS Supervisor that is composed by QoS Monitor (QM) agents, existing as many as QoS to be handled, and one Diagnosis & Decision (D&D) agent. Jointly, they supervise QoS fulfilment and launch diagnosis and decision algorithms when needed.

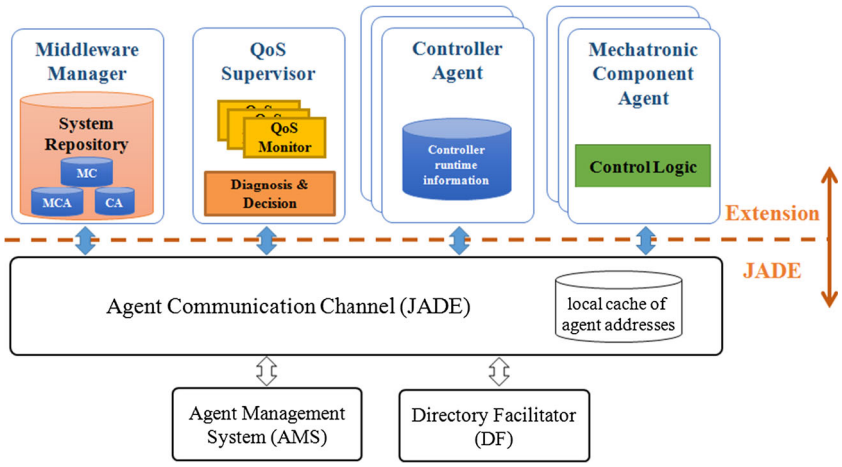


Fig. 1 Middleware implementation

These latter comprises the basic MAS-based architecture for a set of QoS. In addition, there exists a set of agents representing the actual manufacturing application, the so-called resource agents:

- There are as many Controller Agents (CA) as number of controllers in the system.
- Every MC in the application has associated as many MC agents (MCA) as number of controllers in the system can run it. At runtime, only one of them, the active MCA, is responsible for controlling the execution of the corresponding MC. It is also in charge of managing MC reconfiguration using the information about the MC formulated in Section 3.

The rest of the section is dedicated to describe the function of each agent, its role in the architecture and the interaction ontology.

4.1 Middleware Manager

The MM is responsible for maintaining the System Repository that contains the current state of the automation system: current active controllers and active MCAs. This is performed through get_info and set_info ontology commands. This information is updated and read by the supervisor agents. At system start-up, the model of the automation system is registered: controllers characterized by resources (MCs and CA can run and their corresponding MCAs, memory, CPU factor with respect to the reference controller, etc.) and the set of MCs in the whole automation system. Figure 2 depicts the SR meta-model.

4.2 QoS Supervisor

The supervisor is responsible for tracking a set of system QoS, detecting QoS losses and making decisions in order to recover the lost QoS as soon as possible. Its functionality is performed by a set of QM agents and one D&D agent.

4.2.1 QoS Monitor (QM) agents

Each QM is responsible for monitoring specific QoS meeting. This functionality is performed by means of the collaboration between QM and resource agents (MCAs and CAs). Resource agents are in charge of detecting situations that can lead to QoS loss (for instance, loss of controller heartbeat in Availability or maximum workload overtaken in case of system efficiency) and inform to the corresponding QM accordingly. Upon the reception of QoS loss events, the QM sends confirmation requests to avoid false positives, registers the type of QoS lost to handle subsequent events of the same type and launches reconfiguration events.

4.2.2 Diagnosis & Decision (D&D) agent

The D&D is responsible for recovering QoS if it is possible. It receives reconfiguration events from QMs. As multiple reconfiguration events corresponding to different QoS can be issued, D&D analyses the global system state and makes decision to recover QoS by priority. Each QoS requires specific analysis but reconfiguration actions always involved the activation of MCs in other controllers, sometimes after de-activation of MCs and/or execution of recovery actions, depending on the situation.

To perform analysis processes, the D&D involves resource agents (CAs and MCAs) while to make a decision, it launches negotiation processes to get the CA in which a MC will be activated. Finally, it uses diagnosis processes performed by MCAs to launch the reconfiguration or stop the MC warning the operator in case of severe errors.

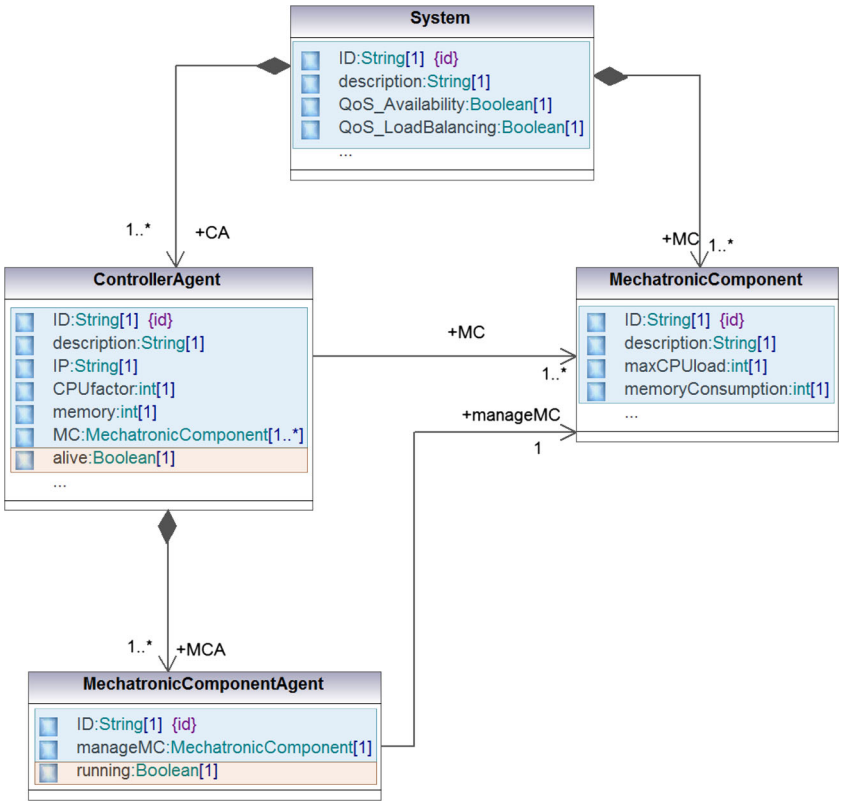


Fig. 2 Middleware Manager System Repository

4.3 Controller Agent

When a controller joins the system, the corresponding Controller Agent (CA) is launched, and it registers the controller and associated resources in the SR. It also registers itself in the DF offering as services the set of MCs that can run in the controller. Finally, it launches an MC (MCA) for each MC.

As part of its functionality, CAs may perform QoS monitoring functions. For instance, for system efficiency QoS, the CA can monitor the current workload of the controller, issuing QoS loss when the workload is less than the minimum or greater than the maximum allowed. It may also be involved in negotiation processes launched by D&D.

4.4 Mechatronic Component Agent

MCAs are in charge of managing the execution of the corresponding MC control logic, which is actually executing in the runtime virtual machine of the controller. Each MCA is able to activate/de-activate the MC, launch the execution of recovery actions as well as collect, transmit, store and make diagnosis on the current state of the MC. MCA operation is defined by the finite state machine (FSM) depicted in Fig. 3. States correspond to JADE agent behaviours and transitions between states are triggered by the D&D as a result of a decision.

- *Boot* state*:* It corresponds to the start-up where initialization actions are performed as well as self-registration in the SR.
- *Active* state*:* While in this state, the MC is executed in the PLC. The state is collected periodically from the PLC runtime and transmitted to the rest of MCAs that are tracking the active. Note that only one MCA of the set corresponding to each MC can be in the active state.

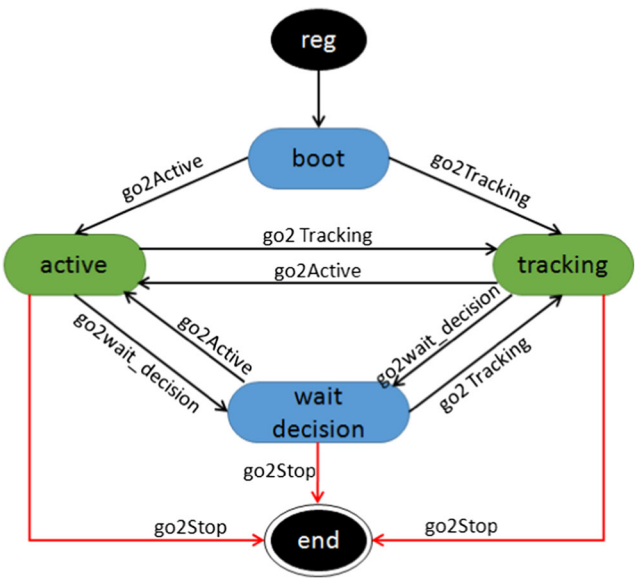


Fig. 3 MCA finite state machine (FSM)

- *Tracking* state*:* While in this state, the MCA receives and stores the current state of the MC, sent by the active MCA.
- *Wait decision* state*:* During the reconfiguration process, the MCA remains in this state, and if required by the D&D, it can perform a diagnosis of the state.
- *End* state*:* When the MCA enters this state, after clean-up tasks are executed, the agent is removed.

Note that the functionality executed in every state by an MCA may be extended to handle different types of QoS. For instance, while in the tracking state, MCAs detect a possible failure of the active MCA if the state is not received within a timeout.

4.5 QoS management ontology

Agent communication is performed through messages following specific syntax and semantics known by the sender and the receiver. The specific messages defined for QoS management form the proposed ontology.

Three phases can be distinguished for assuring QoS: monitoring, loss detection and QoS recovering phases. All three are performed by means of ontology messages exchange.

To illustrate the ontology, let us take, without losing generality as every QoS is monitored, state is diagnosed and decision is made, the case of Availability QoS. Availability implies the service continuity or service recovery as soon as possible and transparently to the application. The following sub-sections illustrate the middleware operation in case of controller failure from Availability monitoring, loss detection under a controller failure up to recovering its service on other controllers. Customized sequence diagrams (to include timeouts) are used to show the messages exchange and the agents involved in each phase.

4.6 QoS monitoring

As commented above, depending on the QoS to be handled, there are application agents (CAs, MCAs) responsible for monitoring QoS meeting and issuing QoS loss events to the QM. In the case of service availability, MCAs in tracking state are responsible for availability monitoring. They receive the current state of the MC from the active MCA. The reception of the state can be used as MCA heartbeat. If the state is not received within a timeout, tracking MCAs issue Availability QoS loss to QM through *QoS_Loss_Event* message. The message contains the situation detected (in this case, MC service unavailability) and the affected resource (MC1). Figure 4 illustrates this situation.

4.6.1 QoS loss detection

Upon the reception of the Availability loss event, QM registers MCA failure to avoid multiple reactions to the same event. Then, QM confirms the QoS loss issuing a confirmation message to the CA. In case the loss of controller is confirmed, the failure is registered to avoid reacting to several events triggered due to other MCs in failure. Next, a reconfiguration event is issued to the D&D through the *QoS_Reconfiguration_Event*, characterized by the QoS affected (availability), type of reconfiguration (controller service recovery) and the agent involved (CA1). Figure 5 illustrates this situation.

4.6.2 Diagnosis, decision and recovery actions

The D&D resolves concurrent reconfiguration events corresponding to different QoS by priority (for instance, service availability has higher priority than system efficiency). If it corresponds to controller failure, it gets from the SR of the MCs to be recovered and issues messages to all their MCAs to provoke a state transition to the *wait_for_decision* state. Next, it launches a negotiation process to all CAs containing MCs to be recovered. Once the negotiation finishes, D&D requires the diagnosis of MCs state to make the correct decision on the recovery. This is illustrated in Fig. 6.

5 Proof of concept: assuring service continuity in scale model of a manufacturing process

This section presents the assessment of the proposed architecture for Availability QoS. In particular, the runtime performance of the proposed architecture implementation is assessed in case of controller failure.

The supervision architecture has been applied to the flexible assembly cell FMS-200, located in the Department of Automatic Control and System Engineering of the Basque

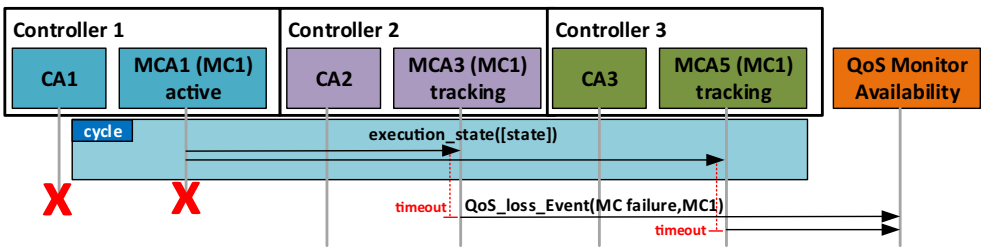


Fig. 4 Availability QoS monitoring and QoS loss detection

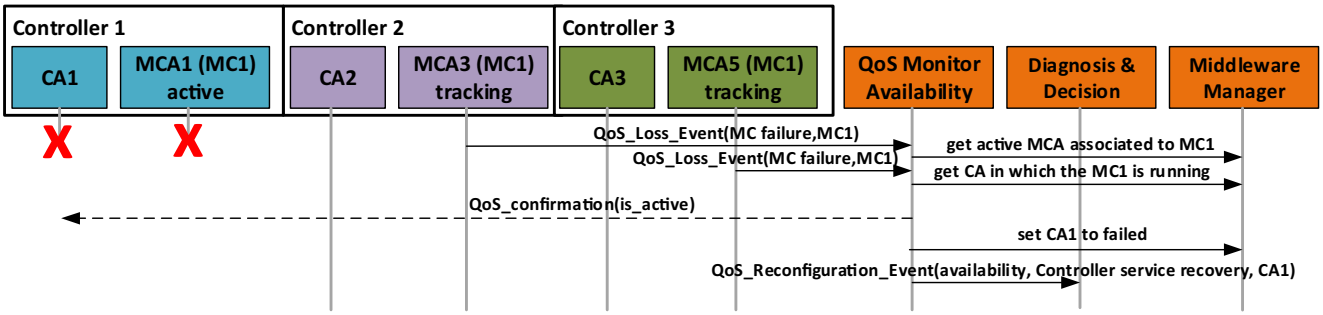


Fig. 5 Availability loss detection

Country University. As it is depicted in the bottom part of the Fig. 7, the cell consists of four stations and a modular conveyor system (Transport Station) that is in charge of assembling a set of four pieces (base, bearing, shaft and lid). In the first station, the base is fed, its orientation is verified and, if correct, it is moved to the pallet located on the transfer system. In the second one, the bearing and shaft are placed on the base, and the cap is put in the third station. In the final station, the set mounted on the pallet is stored in the warehouse.

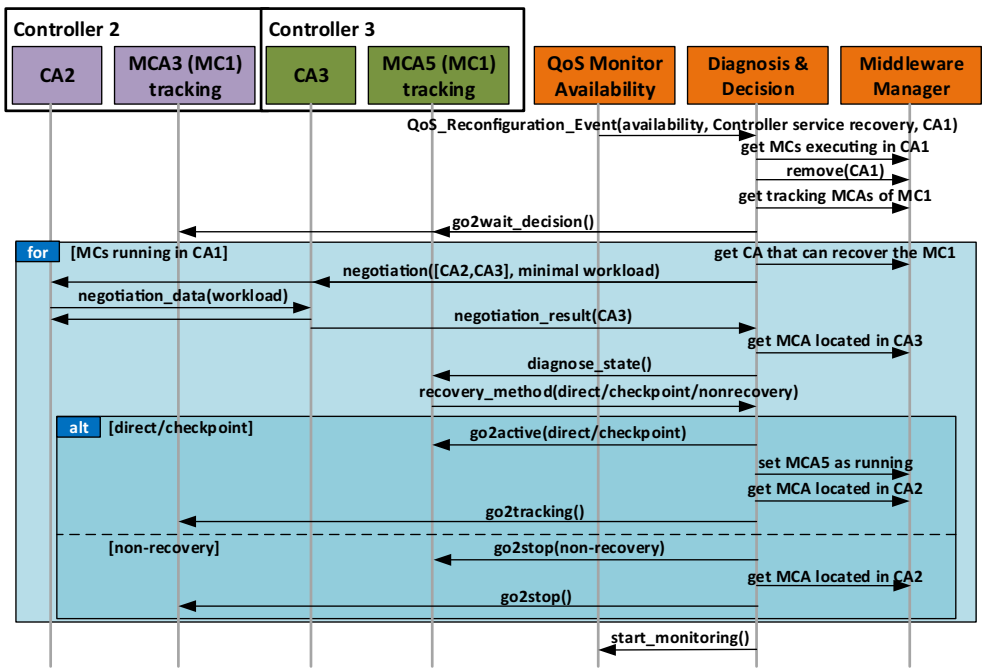
The whole cell is divided into five MCs. For simplicity, the demonstrator is reduced to first and third stations and the mechatronic components associated to them, MC1 and MC2, respectively. The goal of the demonstrator is to show how the architecture manages the failure of a controller in which MC1 or MC2 are being executed (see Fig. 7).

In the first station, the movement of the pneumatic actuator that grippes the base and moves it from the station to the pallet is considered as a critical operation since it must necessarily be executed without interruptions. If it loses the base during this operation, different actions must be taken depending on where the base falls, and the normal way of operation must be restored. Therefore, this leads to two critical intervals defined for MC1:

- A checkpoint recovery if the error occurs during the initial upwards movement of the actuator. An error during this interval requires the base to be removed from the station and a new one to be introduced.
- A non-recoverable one if it occurs during the rest of the movement, from the station to the pallet. An error in this interval causes the base falling into the system and blocking it, so the system must be stopped and the operator should be informed of the failure.

The third station (MC2) inserts lids on the pieces. The different operations undertaken in this station are distributed around an index plate (turning table) leading to multiple sequences executing in parallel. There are different types of lids depending on material, colour and height. To introduce lids into

Fig. 6 Service recovery through negotiation phase



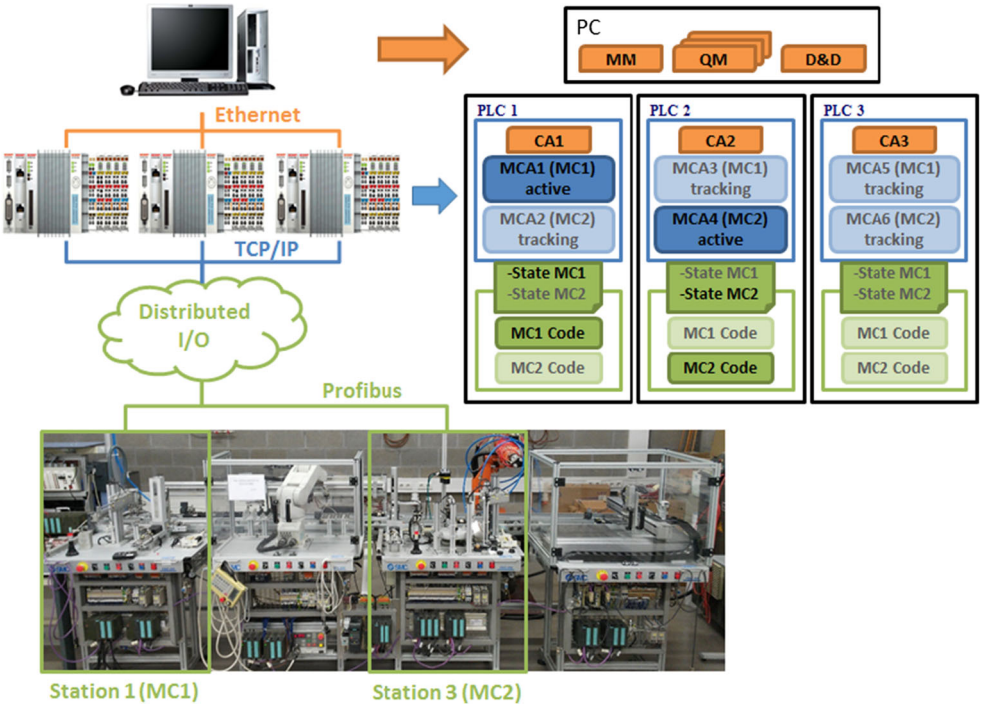


Fig. 7 Manufacturing system demonstrator

the plate, to extract those that not correspond to the current base and to place the lid are, the three, considered as critical operations. Therefore, there will be execution intervals that encompass all the probable parallel execution states of these operations. In fact, there have been identified five critical intervals being all of them recoverable but having different recovery actions to be taken in order to allow the station to continue extracting a new lid from the warehouse. Figure 8 shows these critical intervals for MC1 with its corresponding state values referred as expressions, recovery actions and checkpoints.

As illustrated in Fig. 7, the demonstrator is composed of three Beckhoff CX1020 controllers, in which its respective CA is executed. They may run the PLC code for the two MCs. These controllers are soft PLCs, which run a Windows XP-embedded operating system in parallel with the Beckhoff PLC runtime. The Controller Agents (CAs) and Mechatronic Component Agents (MCAs) associated to MCs will execute in the operating system, while the control code of MCs will execute in the PLC runtime. The demonstrator is also equipped with a PC in which the Middleware Manager and the QoS Supervisor will run (see Fig. 7).

The performance of middleware architecture has been assessed measuring the time of a direct recovery reconfiguration. The measurement is the time interval from the first QoS loss event received by the QM up to the activation of all active MCs in other controllers.

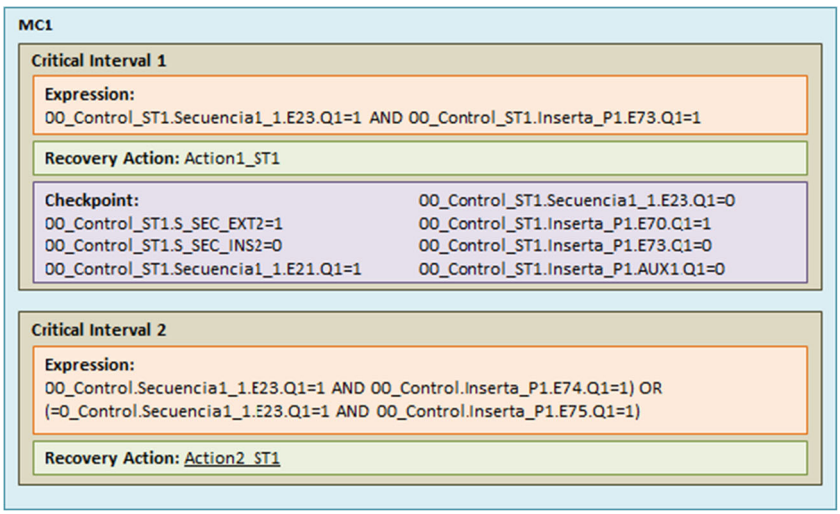


Fig. 8 Critical intervals for the MC1

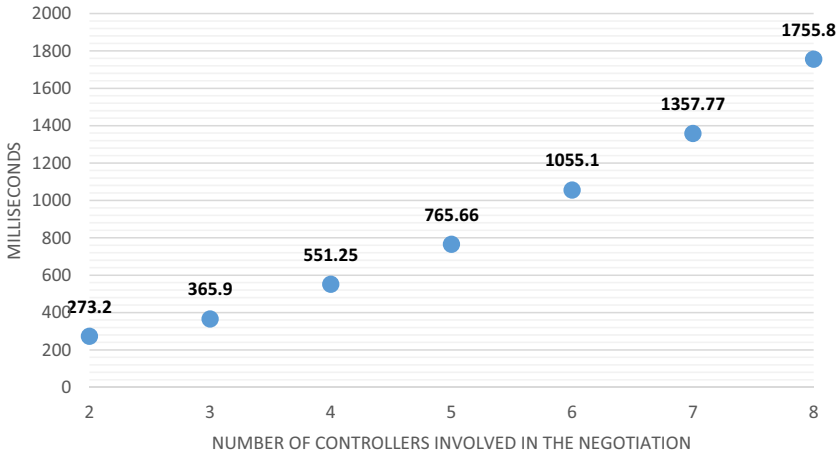


Fig. 9 Recovery time vs. number of controllers involved in the negotiation

The first test performs a comparison of the recovery time based on the number of controllers involved in the negotiation. The result is shown in Fig. 9. As expected, as the number of controllers involved in the negotiation, the time the QM takes to make a decision increases. Thus, there is a trade-off between the redundancy level and the QoS recovery time. It is important to select a number of tracking MCAs that lead to a reasonable recovery time.

The second test compares the recovery time based on the number of MCs to be recovered. The result of this test is presented in Fig. 10. As anticipated, the time increases proportionally to the number of the MCs, illustrating the scalability of the recovery process. Thus, this may help to decide an acceptable granularity in the automation system design, i.e. the number of MCs in which the system should be divided.

6 Conclusions and future work

This paper presents a customizable and extensible architecture able to assure the fulfilment of the quality of service an automation system should offer. The basic mechanism is to reconfigure the system architecture by replicating the control code of a part of the process (a mechatronic component) in different controllers. The reconfiguration consists of making a decision of which controller is in charge of controlling each MC. The proposed architecture consists of a series of application agents that are making use of an ontology monitor QoS, detecting QoS and triggering events. The basic architecture offers a set of agents to resolve multiple events and false positives and recover the QoS if it is possible.

Guidelines to include other QoS include defining monitoring functions using resource agents as well as recovery actions. Besides, information exchange through ontology commands has to be also defined. If necessary, the ontology could be extended with new commands. Thus, the architecture can be instanced to include different monitoring and diagnosis functions extending the agent's functionality. The cost of these mechanisms shows that it is necessary to find a compromise between the quality of the automation services and the overhead it adds. For the case of service availability, this cost has been measured under different redundancies and different numbers of controllers.

From the assessment performed in the case study, it can be concluded that the architecture introduces an overhead that depends very much on design decisions: number of replicas

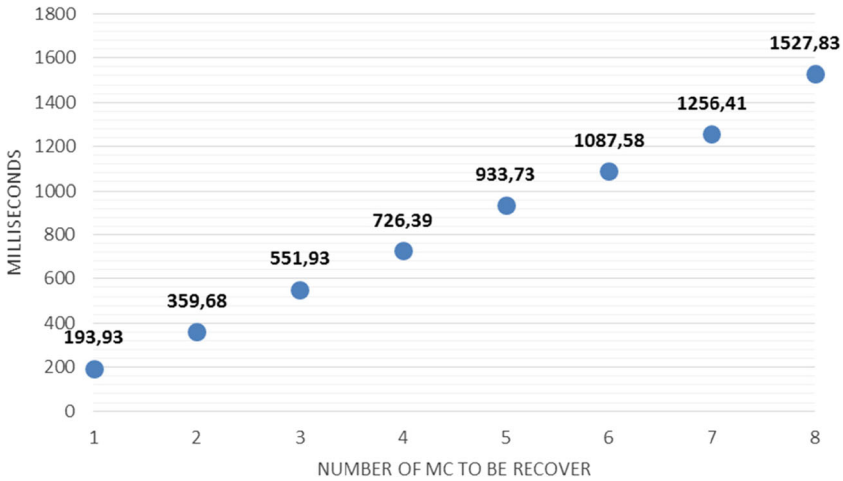


Fig. 10 Recovery time vs. number of MCs to be recovered

for MC and number of controllers containing replicas in the system. Thus, design decisions must be lead by the critical parts of the system, the reliability of the hardware used on the different parts and the level of quality assurance needed in the different parts of the manufacturing process.

Current work focuses on using model-based techniques for achieving automatic code generation for both, agents and PLC code.

Acknowledgments This work was financed by the MINECO/FEDER under project DPI2015-68602-R.

References

1. Innovation ERC (2013) Factories of the Future PPP: towards competitive EU manufacturing. European Union Publishing. http://ec.europa.eu/research/press/2013/pdf/ppp/fof_factsheet.pdf. Accessed 28 Sept 2016
2. Blanchet M, Rinn T, Von Thaden G, Thieulloy G (2014) Industry 4.0 The new industrial revolution How Europe will succeed. Roland Berger Publishing. https://www.rolandberger.com/publications/publication_pdf/roland_berger_tab_industry_4_0_20140403.pdf. Accessed 29 Sept 2016
3. Council NST (2016) Advanced manufacturing : a snapshot of priority technology areas across the federal government subcommittee for advanced manufacturing. White House Publishing. https://www.whitehouse.gov/sites/whitehouse.gov/files/images/Blog/NSTC%20SAM%20technology%20areas%20snapshot.pdf. Accessed 29 Sept 2016
4. Morenas J, Higuera AG, Alonso PG (2012) Product driven distributed control system for an experimental logistics centre. International Journal of Innovative Computing, Informatics and Control 8:7199–7216
5. Nouri H (2015) Development of a comprehensive model and BFO algorithm for a dynamic cellular manufacturing system. Appl Math Model 40:1514–1531. doi:10.1016/j.apm.2015.09.004
6. Pach C, Berger T, Sallez Y, Trentesaux D (2015) Reactive control of overall power consumption in flexible manufacturing systems scheduling: a potential fields model. Control Eng Pract 44:193–208. doi:10.1016/j.conengprac.2015.08.003
7. Lepuschitz W, Zoitl A, Vallée M, Merdan M (2011) Toward self-reconfiguration of manufacturing systems using automation agents. IEEE Transactions on Systems, Man, and Cybernetics, Part C 41:52–69. doi:10.1109/TSMCC.2010.2059012
8. Legat C, Vogel-heuser B (2014) A multi-agent architecture for compensating unforeseen failures on field control level. Service Orientation in Holonic and Multi-Agent Manufacturing and Robotics 544:195–208. doi:10.1007/978-3-319-04735-5
9. Botygin IA, Tartakovsky VA (2014) The development and simulation research of load balancing algorithm in network infrastructures. In: Proceedings of the International Conference on Mechanical Engineering, Automation and Control Systems (MEACS), Tomsk. doi:10.1109/MEACS.2014.6986904
10. Binotto APD, Wehrmeister MA, Kuijper A, Pereira CE (2013) Sm@rtConfig: a context-aware runtime and tuning system using an aspect-oriented approach for data intensive engineering applications. Control Eng Pract 21:204–217. doi:10.1016/j.conengprac.2012.10.001
11. Guo L, Wang B, Wang W (2009) Research of energy-efficiency algorithm based on on-demand load balancing for wireless sensor networks. Proceedings of the International Symposium on Test and Measurement 2:22–26. doi:10.1109/ICTM.2009.5413071
12. Merz M, Frank T, Vogel-Heuser B (2012) Dynamic redeployment of control software in distributed industrial automation systems during runtime. In: Proceedings of the IEEE International Conference on Automation Science and Engineering (CASE), Seoul, pp 863–868. doi:10.1109/CoASE.2012.6386445
13. Streit A, Rösch S, Vogel-Heuser B (2014) Redeployment of control software during runtime for modular automation systems taking real-time and distributed I/O into consideration. In: Proceedings of the IEEE 19th Conference on Emerging Technologies Factory Automation (ETFA). Barcelona. doi:10.1109/ETFA.2014.7005263
14. Priego R, Armentia A, Orive D, Marcos M (2013) Supervision-based reconfiguration of industrial control systems. In: Proceedings of the IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA). Cagliari. doi:10.1109/ETFA.2013.6648130
15. Priego R, Agirre A, Estévez E, Orive D, Marcos M (2015) Middleware-based support for reconfigurable mechatronic systems, 2nd conference on embedded systems. Computational Intelligence and Telematics in Control (CESCIT) 48:81–86. doi:10.1016/j.ifacol.2015.08.112
16. Priego R, Armentia A, Estevez E, Marcos M (2015) On applying MDE for generating reconfigurable automation systems. In: Proceedings of the 13th IEEE International Conference on Industrial Informatics (INDIN). Cambridge, pp 1233–1238. doi:10.1109/INDIN.2015.7281911
17. Commission IE (2004) International Standard IEC 61499 Part 1. IEC Publishing. https://webstore.iec.ch/p-preview/info_iec61499-1%7Bed1.0%7Den.pdf. Accessed 28 Sept 2016
18. Commission IE (2013) International Standard IEC 61131-3, Programmable Logic Controllers Part 3. IEC Publishing. https://webstore.iec.ch/publication/4552. Accessed 28 Sept 2016
19. Olsen S, Wang J, Ramirez-Serrano A, Brennan RW (2005) Contingencies-based reconfiguration of distributed factory automation. Robot Comput Integr Manuf 21:379–390. doi:10.1016/j.rcim.2004.11.011
20. Khalgui M, Mosbahi O (2010) Intelligent distributed control systems. Inf Softw Technol 52:1259–1271. doi:10.1016/j.infsof.2010.06.001
21. Schimmel A, Zoitl A (2011) Distributed online change for IEC 61499. In: Proceedings of the IEEE 16th Conference on Emerging Technologies & Factory Automation (ETFA). Toulouse. doi:10.1109/ETFA.2011.6059117
22. Yan J, Vyatkin V (2013) Extension of reconfigurability provisions in IEC 61499. In: Proceedings of the IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA). Cagliari. doi:10.1109/ETFA.2013.6648026
23. Strasser T, Froschauer R (2012) Autonomous application recovery in distributed intelligent automation and control systems. IEEE Transactions on Systems, Man, and Cybernetics, Part C 42:1054–1070. doi:10.1109/TSMCC.2012.2185928
24. Wehrmeister MA, de Freitas EP, Binotto APD, Pereira CE (2014) Combining aspects and object-orientation in model-driven engineering for distributed industrial mechatronics systems. Mechatronics 24:844–865. doi:10.1016/j.mechatronics.2013.12.008
25. Armentia A, Gangoiti U, Priego R, Estévez E, Marcos M (2015) Flexibility support for homecare applications based on models and multi-agent technology. Sensors 15:31939–31964. doi:10.3390/s151229899

26. Thramboulidis K (2005) Model integrated mechatronics—towards a new paradigm in the development of manufacturing systems. IEEE Transactions on Industrial Informatics 1:54–61
27. Lüder A, Estévez E, Hundt L, Marcos M (2010) Automatic transformation of logic models within engineering of embedded mechatronical units. Int J Adv Manuf Technol 54:1077–1089. doi:10.1007/s00170-010-3010-y
28. Bellifemine F, Poggi A, Rimassa G (2001) Developing multi-agent systems with a FIPA-compliant agent framework. Software - Practice and Experience 31:103–128
29. Bellifemine F, Caire G, Poggi A, Rimassa G (2008) JADE: a software framework for developing multi-agent applications. Lessons learned Information and Software Technology 50:10–21. doi:10.1016/j.infsof.2007.10.008
30. FIPA (2015) Standard FIPA Specifications. FIPA Publishing. http://www.fipa.org/repository/standardspecs.html. Accessed 7 Sept 2015